A Simple Model of Communication APIs – Application to Dynamic Partial-order Reduction

Cristian Rosa, Stephan Merz, and Martin Quinson

INRIA & Nancy University, Nancy, France

Abstract. We are interested in the verification, using model checking, of distributed programs that communicate asynchronously over standard communication APIs such as MPI. This is feasible only if the set of executions that the model checker explores is aggressively reduced to a subset of representative executions, using techniques such as dynamic partial-order reduction. We propose a small set of core primitives in terms of which such APIs can be defined and formally specify these primitives in TLA$^+$. From this specification we derive theorems about the (in)dependence of invocations of the primitives, and use them in a DPOR-based verifier that runs within SimGrid, a simulation framework for distributed programming. Our preliminary experimental results indicate that we obtain good reductions, even though complex network operations are implemented in terms of the core communication primitives.

1 Introduction

Distributed systems are in the mainstream of information technology. They form the core of many important business and scientific applications, as multiple distributed entities may work simultaneously at multiple parts of a problem. Such decomposition may improve greatly the performance of the application, help tolerate component failures or handle problems too large to fit in a single processing unit.

The design of algorithms adapted to this context is particularly difficult: beyond the standard problems of parallel programming, such as race conditions, deadlocks or livelocks, distributed algorithms exhibit specific issues due to the fact that communication between distant nodes is asynchronous. It is therefore difficult to define and maintain coherent notions of global state and global time in distributed systems. Moreover, turning the resulting algorithms into efficient applications also constitutes a challenge since the programmer has to rely on specific programming interfaces presenting complex semantics. A slight change in the underlying semantics may result in drastic changes to the properties the overall system exhibits [2].

This work has been partially funded by ANR 08 SEGI 022

In recent years there has been growing interest into applying space exploration techniques for the analysis and verification of distributed systems. Standard model checkers require a user to represent a distributed application in their input languages, which may lead to higher-level models that are easier to verify, but whose correspondence with the original application can be hard to validate. Some recent tools allow a user to verify the software implementations directly. Examples of such tools include ISP [10], for MPI programs, and MaceMC [6], for programs written in the Mace language.

State space exploration is based on the systematic search of all the possible interleavings of concurrent actions in the application. Because of the exponential number of interleavings, it is mandatory to apply reduction techniques that avoid the exploration of equivalent traces as much as possible. The lack of global state, which constitutes one of the major issue when designing distributed systems, can here turn into an advantage. In fact, most properties of distributed systems depend only on the order of message exchanges, whereas the order of local events of distributed processes is usually irrelevant. In particular, dynamic partial-order reduction (DPOR) has proved to be highly effective when model-checking software [3,6,10].

DPOR requires determining whether two given actions are dependent or not, that is, if the order in which they are executed results in different global states. Dependency is usually approximated: false positives result in redundant exploration, whereas false negatives result in incomplete exploration and hence potentially missed errors. Determining whether actions are dependent requires a clearly specified semantics of the programs that are analyzed, and in particular of the communication API. Usually, APIs are specified semi-formally or informally, lacking the level of precision required to reason about dependency. Pervez et al. [10] formalize a subset of the MPI interface by a TLA$^+$ specification of more than 100 pages.

Communication APIs are often complex because they offer operations that could be simulated by more elementary ones but for which it is important to offer optimized implementations. This is particularly true for collective operations. For example, the `MPI_All2All` operation performs global data exchange between many nodes. If a formal model of the standard reflects such operations, which occur prominently in MPI, the dependency analysis becomes even more complicated.

For the design of the future non-blocking collective operations envisioned for MPI-3, Hoefler et al. [5] introduced the Group Operation Assembly Language (GOAL). This is a domain specific language in which group operation algorithms are described as the composition of simpler communication primitives and local data transformation. The purpose of the formalism is to help cope with the complexity of the resulting communication patterns.

We believe that a similar approach is even more relevant for formal analysis because at the semantic level the complex operations are faithfully described as a composition of more elementary ones. In particular, we only need to imple-

ment DPOR for a core of elementary primitives and express the more complex operations in terms of this core set.

In this article we present such a core set of networking primitives, which is at the basis of the communication kernel of the SimGrid [1] simulation framework. SimGrid is a distributed system simulator for heterogeneous platforms that provides several communication interfaces, each tailored for different modeling needs. The set of core operations we introduce is sufficient to express all of them, including a subset of MPI called SMPI, a mailbox based communication interface called MSG and a socket oriented API called GRAS. We formally specify the primitives in TLA$^+$, and we prove independency of certain primitives from our formal specification. We have implemented a state space exploration algorithm in SimGrid that relies on DPOR, according to the independency relations proved before. We use this algorithm to perform reachability analysis on distributed programs executed by SimGrid.

To our knowledge this is the first attempt to determine a reduced set of primitives that simplify the application of DPOR to multiple communication APIs. Pervez et al. [10] apply DPOR to verify practical MPI programs, but unless our work, besides targetting only MPI, they assume that the programs to verify are correct according to the standard, and in particular that they do not access the buffers used in asynchronous communication until a call to `Wait` returns or a call to `Test` returns true. This assumption is in part required because of the way the tool is implemented: they intercept all the relevant MPI calls, decide a schedule, and issue them in a deferred way to the MPI runtime. If the buffers could be accessed unrestrictedly the behaviour of the MPI runtime would be unspecified. Instead, in SimGrid the MPI runtime is emulated on top of the networking primitives presented in this article and thus unauthorized accesses to the buffers can be handled correctly and reported to the user. Yang et al. [11] also use DPOR but focus on distribued systems that use WinAPI and that are deployed on the real execution platform.

The rest of this article is organized as follows: section 2 introduces the networking primitives and their formal specification, section 3 presents the DPOR exploration algorithm and theorems about dependency of primitives based on the framework introduced in section 2, and finally section 4 show some experimental results obtained using SimGrid.

2 Formal Specification of Network Primitives

In this section we present the formal semantics of the communication primitives on top of which the higher level APIs are expressed. The communication model is based on the concept of mailbox or rendez-vous points. Processes willing to communicate post their send or receive requests into mailboxes that queue them. Actual communication starts when a matching request is posted. Our set of core network primitives contains just four operations: *Send*, *Recv*, *WaitAny*, and *Test*. The first two post a send (resp., receive) request to a mailbox. *WaitAny*

```
MODULE SimixNetwork
EXTENDS Naturals, Sequences, FiniteSets
CONSTANTS RdV, Addr, Proc, Program, ValTrue, ValFalse,
          SendIns, RecvIns, WaitIns, TestIns, LocalIns
Partition(S) ≜ ∀x, y ∈ S : x ∩ y = {} ∨ x = y
Instr ≜ UNION {SendIns, RecvIns, WaitIns, TestIns, LocalIns}
NoProc ≜ CHOOSE p : p ∉ Proc
NoAddr ≜ CHOOSE a : a ∉ Addr
ASSUME ValTrue ∈ Nat ∧ ValFalse ∈ Nat
ASSUME Partition({SendIns, RecvIns, WaitIns, TestIns, LocalIns})
ASSUME Program ∈ [Proc → Seq(Instr)]
VARIABLES net, mem, pc
---
Comm ≜ [id : Nat, rdv : RdV,
        status : {"send", "recv", "ready", "done"}
        src : Proc, dst : Proc,
        data_src : Addr, data_dst : Addr]
mailbox(rdv) ≜ {comm ∈ net :
                 comm.rdv = rdv ∧ comm.status ∈ {"send", "recv"}
CommBuffers(pid) ≜
    {c.data_src : c ∈ {d ∈ net : d.status ≠ "done" ∧ (d.src = pid ∨ d.dst = pid)}
  ∪ {c.data_dst : c ∈ {d ∈ net : d.status ≠ "done" ∧ (d.src = pid ∨ d.dst = pid)}
TypeInvariant ≜
  ∧ net ⊆ Comm
  ∧ mem ∈ [Proc → [Addr → Nat]]
  ∧ pc ∈ {f ∈ [Proc → Nat] : ∀p ∈ Proc : f[p] ∈ 1..Len(Program[p])}
```

Fig. 1: TLA$^+$ model of the communication network: data model.

allows a process to wait for the completion of any one among a set of communication requests, and *Test* checks if a given communication request has been completed.

We now present a formal specification of the network model, written in TLA$^+$ [8]. We also give evidence that full-fledged communication APIs can be faithfully represented in this model.

2.1 Overall Network Model

The first part of our TLA$^+$ model, showing the data model of the network, appears in Fig. 1. We model the network itself as well as an abstraction of a distributed program that uses it: this allows us to consider the invocation of network operations by the program and prove independence results between these invocations. The model is based on parameters *RdV*, *Addr*, *Proc*, and *Program*, which represent the set of rendez-vous points, memory addresses, processes, and the

program. A multi-process program is represented as an array (indexed by processes) of finite instruction sequences. We distinguish between instructions that invoke network operations (send, receive, wait, and test) and local instructions; these sets are assumed to be pairwise disjoint. For future use, we also introduce "null" values *NoProc* and *NoAddr* for processes and addresses.

The system state is represented by three state variables *net*, *mem*, and *pc*. The variable *net* holds the history of (pending or completed) communication requests (in *Comm*), which are modeled as records containing a request identifier, the rendez-vous point, the status of the request, the source and destination processes, as well as the memory addresses from which the message content will be taken or where it will be delivered. Variable *mem* represents the current memory contents per process, and *pc* points to the instruction that will be executed next, for each process.

Next, the module introduces two operators: $mailbox(rdv)$ collects the pending requests for a given rendez-vous point, and $CommBuffers(pid)$ is the set of memory addresses that appear in communication requests involving process *pid*, which have not yet completed.

Since TLA$^+$ is untyped, we document the intended types of the state variables by a type invariant, which will have to be shown to be preserved by each possible transition. The network is a set of communications, the system memory is modeled as an nested array associating processes and addresses to natural numbers (representing memory values). Finally, the program counters are modeled as an array yielding for each process some index that points to an instruction of that process.

2.2 Communication Primitives

Figure 2 shows specifications of the primitive operations for our network model. Process *pid* can post a *Send* request for mailbox *rdv* when its program counter is at a "send" instruction. We distinguish two cases: if a receive request is waiting for communication at *rdv*, then the send request is matched with the oldest (lowest-numbered) such receive request. The status of the request changes to "ready", indicating that the communication can now actually be performed, and the *src* and *data_src* fields are updated according to the parameters of the send request. The communication ID is stored at the memory address indicated by process *pid*.

If no pending receive request exists for *rdv*, then a new communication record for *rdv* is created in the network from the parameters of the send request. The status of this request is set to "send", indicating that it is waiting for a matching receive request, and its ID is stored in the memory of process *pid*. In either case, the program of process *pid* advances to some new instruction. (Remember that this is an abstract program model and that any concrete program should be a refinement of what is allowed by this specification.)

The axioms of set theory ensure that for any set there exists some value that is not an element of that set.

$$
\begin{array}{l}
\textit{Send}(\textit{pid}, \textit{rdv}, \textit{data\_r}, \textit{comm\_r}) \triangleq \\
\quad \land\ \textit{Program}[\textit{pid}][\textit{pc}[\textit{pid}]] \in \textit{SendIns} \\
\quad \land\ \lor\ \exists c \in \textit{mailbox}(\textit{rdv}) : \\
\qquad\quad \land\ c.\textit{status} = \text{“recv”} \\
\qquad\quad \land\ \forall d \in \textit{mailbox}(\textit{rdv}) : d.\textit{status} = \text{“recv”} \Rightarrow c.\textit{id} \leq d.\textit{id} \\
\qquad\quad \land\ \textit{net}' = (\textit{net} \setminus \{c\}) \cup \{[c \text{ EXCEPT } !.\textit{status} = \text{“ready”}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad !.\textit{src} = \textit{pid}, !.\textit{data\_src} = \textit{data\_r}]\} \\
\qquad\quad \land\ \textit{mem}' = [\textit{mem} \text{ EXCEPT } ![\textit{pid}][\textit{comm\_r}] = c.\textit{id}] \\
\qquad \lor\ \land\ \neg\exists c \in \textit{mailbox}(\textit{rdv}) : c.\textit{status} = \text{“recv”} \\
\qquad\quad \land\ \text{LET } \textit{comm} \triangleq [\textit{id} \mapsto \textit{Cardinality}(\textit{net}) + 1, \\
\qquad\qquad\qquad\qquad\qquad\quad \textit{rdv} \mapsto \textit{rdv},\ \ \textit{status} \mapsto \text{“send”}, \\
\qquad\qquad\qquad\qquad\qquad\quad \textit{src} \mapsto \textit{pid},\ \ \textit{dst} \mapsto \textit{NoProc}, \\
\qquad\qquad\qquad\qquad\qquad\quad \textit{data\_src} \mapsto \textit{data\_r},\ \ \textit{data\_dst} \mapsto \textit{NoAddr}] \\
\qquad\qquad \text{IN} \quad \land\ \textit{net}' = \textit{net} \cup \{\textit{comm}\} \\
\qquad\qquad\qquad\quad \land\ \textit{mem}' = [\textit{mem} \text{ EXCEPT } ![\textit{pid}][\textit{comm\_r}] = \textit{comm.id}] \\
\quad \land\ \exists \textit{pos} \in 1..\textit{Len}(\textit{Program}[\textit{pid}]) : \textit{pc}' = [\textit{pc} \text{ EXCEPT } ![\textit{pid}] = \textit{pos}] \\
\textit{Recv}(\textit{pid}, \textit{rdv}, \textit{data\_r}, \textit{comm\_r}) \triangleq \ \ldots \\
\textit{Wait}(\textit{pid}, \textit{comms}) \triangleq \\
\quad \land\ \textit{Program}[\textit{pid}][\textit{pc}[\textit{pid}]] \in \textit{WaitIns} \\
\quad \land\ \exists \textit{comm} \in \textit{comms}, c \in \textit{net} : c.\textit{id} = \textit{mem}[\textit{pid}][\textit{comm}] \land \\
\qquad \lor\ \land\ c.\textit{status} = \text{“ready”} \\
\qquad\quad \land\ \textit{mem}' = [\textit{mem} \text{ EXCEPT } ![c.\textit{dst}][c.\textit{data\_dst}] = \textit{mem}[c.\textit{src}][c.\textit{data\_src}]] \\
\qquad\quad \land\ \textit{net}' = (\textit{net} \setminus \{c\}) \cup \{[c \text{ EXCEPT } !.\textit{status} = \text{“done”}]\} \\
\qquad \lor\ c.\textit{status} = \text{“done”} \land \text{UNCHANGED } \langle \textit{mem}, \textit{net} \rangle \\
\quad \land\ \exists \textit{pos} \in 1..\textit{Len}(\textit{Program}[\textit{pid}]) : \textit{pc}' = [\textit{pc} \text{ EXCEPT } ![\textit{pid}] = \textit{pos}] \\
\textit{Test}(\textit{pid}, \textit{comm}, \textit{ret}) \triangleq \\
\quad \land\ \textit{Program}[\textit{pid}][\textit{pc}[\textit{pid}]] \in \textit{TestIns} \\
\quad \land\ \lor\ \exists c \in \textit{net} : c.\textit{id} = \textit{mem}[\textit{pid}][\textit{comm}] \land \\
\qquad\quad \lor\ \land\ c.\textit{status} = \text{“ready”} \\
\qquad\qquad \land\ \textit{mem}' = [\textit{mem} \text{ EXCEPT } ![c.\textit{dst}][c.\textit{data\_dst}] = \textit{mem}[c.\textit{src}][c.\textit{data\_src}], \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad ![\textit{pid}][\textit{ret}] = \textit{ValTrue}] \\
\qquad\qquad \land\ \textit{net}' = (\textit{net} \setminus \{c\}) \cup \{[c \text{ EXCEPT } !.\textit{status} = \text{“done”}]\} \\
\qquad\quad \lor\ \land\ c.\textit{status} = \text{“done”} \\
\qquad\qquad \land\ \textit{mem}' = [\textit{mem} \text{ EXCEPT } ![\textit{pid}][\textit{ret}] = \textit{ValTrue}] \\
\qquad\qquad \land\ \text{UNCHANGED } \textit{net} \\
\qquad \lor\ \land\ \neg\exists c \in \textit{network} : c.\textit{id} = \textit{memory}[\textit{pid}][\textit{comm}] \land c.\textit{status} \in \{\text{“ready”}, \text{“done”}\} \\
\qquad\quad \land\ \textit{mem}' = [\textit{mem} \text{ EXCEPT } ![\textit{pid}][\textit{ret}] = \textit{ValFalse}] \\
\qquad\quad \land\ \text{UNCHANGED } \textit{net} \\
\quad \land\ \exists \textit{pos} \in 1..\textit{Len}(\textit{Program}[\textit{pid}]) : \textit{pc}' = [\textit{pc} \text{ EXCEPT } ![\textit{pid}] = \textit{pos}] \\
\textit{Local}(\textit{pid}) \triangleq \\
\quad \land\ \textit{Program}[\textit{pid}][\textit{pc}[\textit{pid}]] \in \textit{LocalIns} \\
\quad \land\ \textit{mem}' \in [\textit{Proc} \to [\textit{Addr} \to \textit{Nat}]] \\
\quad \land\ \forall p \in \textit{Proc}, a \in \textit{Addr} : \textit{mem}'[p][a] \neq \textit{mem}[p][a] \\
\qquad\qquad\qquad\qquad\qquad \Rightarrow p = \textit{pid} \land a \notin \textit{CommBuffers}(\textit{pid}) \\
\quad \land\ \exists \textit{pos} \in 1..\textit{Len}(\textit{Program}[\textit{pid}]) : \textit{pc}' = [\textit{pc} \text{ EXCEPT } ![\textit{pid}] = \textit{pos}] \\
\quad \land\ \text{UNCHANGED } \textit{net}
\end{array}
$$

Fig. 2: TLA$^+$ model of the communication network: operations (excerpt).

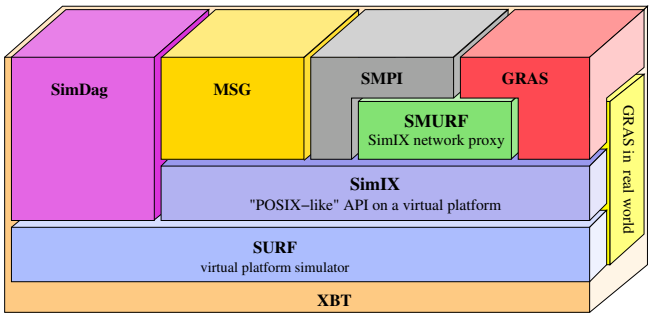

Fig. 3: Architecture of the SimGrid framework.

The specification of the *Recv* operation is symmetrical and omitted from figure 2. A *Wait* operation allows a process to wait for the completion of any of a set *comms* of network operations (represented by the memory addresses in which their communication IDs are stored). It is enabled as soon as the status of one of these operations is either "ready" or "done". If the status is "ready", the communication is performed by transferring the contents of the memory buffer of the source process to the memory buffer of the destination process, and the status is updated to "done". If the status was already "done" before the *Wait* operation, the operation has no effect on the network or the memory.

The operation *Test* can be used by a process to check if a given communication request *comm* has completed or is ready to complete. If so, the primitive acts like a *Wait* for the singleton set $\{comm\}$, but also returns the result *ValTrue* in the memory address indicated by parameter *ret*. Otherwise, it returns *ValFalse*, but does not block the calling process as a *Wait* instruction would.

Finally, we also model local operations of processes by a loosely specified action *Local*, which does not modify the network, but may update the memory of the process that executes the operation, except for any locations that are the source or destination addresses of pending network operations in which the process participates.

The overall next-state action of the specification (not shown in figure 2) is simply defined as the disjunction of these elementary actions, for parameter values ranging over the appropriate sets.

2.3 Expressiveness

The comunication primitives presented in section 2.2 have been implemented as the core of the SimIX networking layer of the SimGrid framework, whose architecture is depicted in figure 3. This allows us to empirically validate the expressiveness of these primitives, with respect to both feasibility (by implementing different communication APIs on top of SimIX) and performance (by executing test suites of these APIs and comparing them to native implementations).

For SMPI, a representative subset of MPI, we use the NAS Parallel Benchmarks suite that executes succesfully in the simulator. However, SMPI does not implement features like one-sided communication.

MSG is a simple mailbox-based communication API. There are more than 50 publications that make use of it to perform experiments with the simulator. Many of the programs of these experiments are part of a regression testing suite for MSG incorporated into the SimGrid framework. Initially this API was not implemented on top of the primitives provided by SimIX, but after being ported the entire test suite still executes correctly.

GRAS is a socket-based communication infraestructure, and it is currently being ported on top of the new networking layer. Although a few tests still fail to execute properly, this is due to technical problems in the implementation and not the networking primitives.

3 Dynamic Partial-Order Reduction

Dynamic partial-order reduction [3,9] is a technique for coping with the state explosion problem by avoiding the exploration of executions that differ only in the order of execution of independent operations. The technique has proved very effective for software model checking. After an introduction of the DPOR algorithm, we derive (in)dependency relations between the primitives introduced in section 2.

3.1 Overview

The idea of the DPOR algorithm is to examine only a representative subset of all possible interleavings of the distributed processes. The algorithm is correct provided every non-explored interleaving is semantically equivalent to at least one interleaving that has been explored, so that no potential error is missed.

To understand how DPOR works, consider the model of a distributed system viewed as a set of *happened-before* relations [7], one for each possible run. Each run defines a partial-order in the set of local states.

The model checker explores global states and thus generates global traces of the system that are possible serializations of these happened-before relations. Because a happened-before relation is a partial order, there can be many serializations that differ only in the execution order of concurrent independent transitions. DPOR tries to explore only one serialization for each partial order, based on information about which transitions are independent.

Precisely determining (in)dependency of transitions can be costly, as it involves evaluating the precise effects of two transitions in either order, and the overhead incurred by this computation may annihilate the benefits of using DPOR. In practice, dependency is therefore approximated, and for soundness this approximation has to be conservative in the sense that two transitions should be considered dependent except when one can prove the contrary. On the other hand, the more conservative the relation is, the less reduction is obtained. A

formal framework like the one presented in section 2 helps justify that two given actions are independent.

Algorithm 1 DPOR-based depth first search

```
 1: q := initial state
 2: s := empty
 3: for some p ∈ Proc that has an enabled transition in q do
 4:    interleave(q) := {p}
 5: end for
 6: push(s,q)
 7: while |s| > 0 do
 8:    q := top(s)
 9:    if |unexplored(interleave(q))| > 0 ∧ |s| < BOUND then
10:       t := nextinterleaved(q)
11:       q' := succ(t, q)
12:       for some p ∈ Proc that has an enabled transition in q' do
13:          interleave(q') := {p}
14:       end for
15:       push(s,q')
16:    else
17:       if ∃ i ∈ dom(s): Depend(tran(s_i), tran(q)) then
18:          j := max({i ∈ dom(s): Depend(tran(s_i), tran(q))})
19:          interleave(s_{j-1}) := interleave(s_{j-1}) ∪ {proc(tran(q))}
20:       end if
21:       pop(s)
22:    end if
23: end while
```

Algorithm 1 shows the DPOR depth first search as implemented in SimGrid. It is a modified, depth-bounded version of the algorithm presented in [9] and is used to detect deadlocks or violations of assertions. The state space is explored up to a predefined depth bound, unless the program terminates earlier. Verification is therefore only partial in general, but the model checker does not have to store visited states, which would be too memory-intensive for verifying distributed programs (as opposed to abstract models).

With each state `q` in the search stack `s` is associated a subset `interleave(q)` of some processes that have enabled transitions from `q`. Processes in this set are scheduled in all permutation orders. When a state is reached for the first time (and hence pushed on the stack), the interleave set is populated with a random process chosen among the enabled ones.

The function `nextinterleaved` iterates over the enabled transitions of the processes in the interleave set, updating information about the already executed ones. When all the transitions of the processes in the interleave set have been explored or the depth bound is reached, the state is popped from the search stack.

When a state `q` is about to be popped, it is checked whether the transition that was executed to generate it, denoted by `tran(q)`, is dependent with any previously executed transition in the current trace. If a (most recent) dependent transition $\texttt{tran(s}_j\texttt{)}$ is found, then the process that executed `tran(q)` is added to the interleave set of the state $\texttt{s}_{j-1}$, from which the dependent transition was executed.

3.2 Justifying (In)dependence of Transitions

Determining (in)dependence between transitions is fundamental for an efficient and correct DPOR-based exploration algorithm. We now formally state and prove independence between the fundamental primitives introduced in section 2.

Two actions A and B are independent [4] if at any state where both are enabled, neither disables the other one, and executing the actions in either order leads to the same state. This can be expressed in TLA$^+$ as the following predicate over actions:

$$\begin{aligned} I(A,B) \triangleq \textsc{Enabled}\ A \land \textsc{Enabled}\ B \Rightarrow &\land A \Rightarrow (\textsc{Enabled}\ B)' \\ &\land B \Rightarrow (\textsc{Enabled}\ A)' \\ &\land A \cdot B \equiv B \cdot A \end{aligned}$$

Observe that by definition, independence is symmetric: $I(A,B) \equiv I(B,A)$. The actions A and B are dependent if $\neg I(A,B)$. Typically, two actions are dependent if they may modify shared parts of the state space, such as shared objects or memory buffers.

Based on the TLA$^+$ specifications of the network primitives, we now state several theorems regarding the independence of transitions.

Theorem 1. *Any two Send and Recv transitions are independent.*

$$\begin{aligned} &\forall p_1, p_2 \in \mathit{Proc}, \mathit{rdv}_1, \mathit{rdv}_2 \in \mathit{RdV}, d_1, d_2 \in \mathit{Addr}, c_1, c_2 \in \mathit{Addr} : \\ &\quad I(\mathit{Send}(p_1, \mathit{rdv}_1, d_1, c_1), \mathit{Recv}(p_2, \mathit{rdv}_2, d_2, c_2)) \end{aligned}$$

Proof (sketch). This result might be surprising at first glance. Assume that the *Send* and *Recv* action are both enabled. Then the processes p_1 and p_2 must be different, since they must be at a send and receive instruction, respectively, and these sets are assumed disjoint. It is easy to see that no transition can disable the other from happening. Moreover, both transitions only modify the network state, since data is only transmitted during *Wait* or *Test* steps. If the *Send* and *Recv* transitions concern different rendez-vous points, then there is no shared state modified and thus they are trivially independent. On the contrary, if the requests are posted for the same rendez-vous point, two situations can happen: the mailbox is empty and the two requests match each other independently in which order they are performed, or there are some pending requests (which must be of the same type, either all *Send* or all *Recv* requests), and the matching transition always pairs with the one with the lowest id, the head of the queue. The

other transition, being of the same type as the requests pending on the rendez-vous point, is added at the tail of the queue. This also happens independently of the order in which the transitions are executed, and the resulting state is the same, which proves the theorem. □

Theorem 2. *Two Send or two Recv operations performed by different processes and posted to different rendez-vous points are independent.*

$$\begin{aligned}\forall p_1, p_2 \in Proc, rdv_1, rdv_2 \in RdV, d_1, d_2 \in Addr, c_1, c_2 \in Addr :\\ p_1 \neq p_2 \wedge rdv_1 \neq rdv_2 \Rightarrow \wedge\ I(Send(p_1, rdv_1, d_1, c_1), Send(p_2, rdv_2, d_2, c_2))\\ \wedge\ I(Recv(p_1, rdv_1, d_1, c_1), Recv(p_2, rdv_2, d_2, c_2))\end{aligned}$$

Proof (sketch). From the definition of the *Send* action, it is easy to see that if the processes and the rendez-vous points are different, the two actions modify disjoint parts of the state space. Moreover, execution of one action will not disable the other one, and therefore the actions are independent.

The proof of independence for two *Recv* action is analogous. □

Theorem 3. *Wait or Test operations for the same communication request are independent.*

$$\begin{aligned}&\forall p_1, p_2 \in Proc, c \in Addr : I(Wait(p_1, \{c\}), Wait(p_2, \{c\}))\\ &\forall p_1, p_2 \in Proc, c, r_1, r_2 \in Addr : I(Test(p_1, c, r_1), Test(p_2, c, r_2))\\ &\forall p_1, p_2 \in Proc, c, r_2 \in Addr : I(Wait(p_1, \{c\}), Test(p_2, c, r_2))\end{aligned}$$

Proof (sketch). Assume that both *Wait* operations are enabled. Execution of the first one changes the status of the communications to "done" and copies the data from the sender to the receiver. The second *Wait* transition then finds the communication with "done" status and leaves the shared state unchanged. Because the memory addresses of the buffers of the sender and receiver are stored in the communication, the order of execution doesn't affect the final state.

The proof of independence of two *Test* actions, or for a *Wait* and a *Test* action, for the same communication request is similar. □

Theorem 4. *Any two local actions of different processes are mutually independent.*

$$\forall p_1, p_2 \in Proc : p_1 \neq p_2 \Rightarrow I(Local(p_1), Local(p_2))$$

Proof (sketch). Local actions of different processes modify disjoint parts of the system state space, hence they obviously commute. □

Theorem 5. *Any two Local and Send or Recv transitions are independent.*

$$\begin{aligned}\forall p_1, p_2 \in Proc, rdv \in RdV, d \in Addr, c \in Addr :\\ \wedge\ I(Local(p_1), Send(p_2, rdv, d, c))\\ \wedge\ I(Local(p_1), Recv(p_2, rdv, d, c))\end{aligned}$$

Proof (sketch). Consider a *Local* and a *Send* action. Again, p_1 and p_2 must be different processes due to the assumption that the sets *SendIns* and *LocalIns* are disjoint. Therefore, they modify disjoint parts of array *mem* (in particular, the assumption on the modifications allowed by *Local* actions implies that only the memory of the process p_1 may be affected), and the only modifications to *net* are due to the *Send* action, and they are independent of any modifications that the *Local* action may perform.

The proof of independence between *Local* and *Recv* actions is analogous. □

Theorem 6. *Any two Local and Wait or Test transitions are independent.*

$$\begin{array}{l}\forall p_1, p_2 \in \mathit{Proc}, \mathit{comm} \in \text{SUBSET } \mathit{Addr}, c, r \in \mathit{Addr} : \\ \quad \land\ I(\mathit{Local}(p_1), \mathit{Wait}(p_2, \mathit{comm})) \\ \quad \land\ I(\mathit{Local}(p_1), \mathit{Test}(p_2, c, r))\end{array}$$

Proof (sketch). Consider a *Local* and a *Wait* action. For the same reasons as before, the processes performing these actions must be distinct and the only possible modification to the network stems from the *Wait* action and is independent of whatever modification the *Local* action performs. In case the *Wait* action modifies a memory location, it concerns the destination process of the communication request that is completed, and a location that is included in the *CommBuffers* of that process. Therefore, the *Local* action is not allowed to modify the same location, and the effects on the memory of the two actions must commute. Moreover, the *Local* action cannot disable the *Wait* action; in particular, it cannot modify the memory addresses in *comm* since the two processes are distinct.

The proof of independence between *Local* and *Test* actions is analogous. □

Using the previous results we can now define an independence predicate

$$\begin{array}{rcl}\mathit{Indep}(t_i, t_j) & \triangleq & \lor\ t_i = \mathit{Send}(\_, \_, \_, \_) \land t_j = \mathit{Recv}(\_, \_, \_, \_) \\ & & \lor\ \land\ t_i = \mathit{Send}(p_1, \mathit{rdv}_1, \_, \_) \land t_j = \mathit{Send}(p_2, \mathit{rdv}_2, \_, \_) \\ & & \quad \land\ p_1 \neq p_2 \land \mathit{rdv}_1 \neq \mathit{rdv}_2 \\ & & \lor\ \land\ t_i = \mathit{Recv}(p_1, \mathit{rdv}_1, \_, \_) \land t_j = \mathit{Recv}(p_2, \mathit{rdv}_2, \_, \_) \\ & & \quad \land\ p_1 \neq p_2 \land \mathit{rdv}_1 \neq \mathit{rdv}_2 \\ & & \lor\ t_i = \mathit{Wait}(\_, \{c\}) \land t_j = \mathit{Wait}(\_, \{c\}) \\ & & \lor\ t_i = \mathit{Test}(\_, c, \_) \land t_j = \mathit{Test}(\_, c, \_) \\ & & \lor\ t_i = \mathit{Wait}(\_, \{c\}) \land t_j = \mathit{Test}(\_, c,) \\ & & \lor\ t_i = \mathit{Local}(\_) \land t_j = \mathit{Local}(\_) \\ & & \lor\ t_i = \mathit{Local}(\_) \land t_j = \mathit{Send}(\_, \_, \_, \_) \\ & & \lor\ t_i = \mathit{Local}(\_) \land t_j = \mathit{Recv}(\_, \_, \_, \_) \\ & & \lor\ t_i = \mathit{Local}(\_) \land t_j = \mathit{Wait}(\_, \_) \\ & & \lor\ t_i = \mathit{Local}(\_) \land t_j = \mathit{Tes}(\_, \_, \_)\end{array}$$

Finally, the dependency relation is defined as

$$\mathit{Depend}(t_i, t_j) \triangleq \neg\big(\mathit{Indep}(t_i, t_j) \lor \mathit{Indep}(t_j, t_i)\big)$$

4 Experiments

In this section we present some experimental results obtained using the model-checker implemented in SimGrid, which is based on the DPOR algorithm shown in section 3.1.

We examined three simple algorithms written in C that use the MSG communication API. For simplicity of exposition, we show the pseudocode versions of the algorithms.

Server code

```
1  count = 0
2  while (count<3) {
3   value = receive_from("mailbox")
4   count++
5  }
6  assert( value == 3 )
```

Code of clients 1..3

```
client(int ID) {
  // do something
  if(done){
    send_to(ID, "mailbox")
  }
}
```

Fig. 4: Code of the First Flawed Example.

Figure 4 shows a first flawed program, where the programmer assumed a given message ordering which may violated in practice. In this example, a server process waits for messages arriving from three client processes. Let us assume that the server process has ID 0 and the clients have IDs between 1 and 3. As expressed by line 6 of the server's code, it is assumed that the messages are received in the order of clients' IDs and that the last received message will be the one of client 3. This is however not true since the order in which the clients are executed is non-deterministic.

In this case the DPOR algorithm using the dependency predicate *Depend* defined in section 3.2 explores 371 traces before finding the counter-example shown on the left-hand side of figure 6. Using a trivial dependency predicate under which any pair of transitions is considered dependent the algorithm explores 504 traces.

We also measured the total number of traces explored to cover the entire state space, without checking the assertion. With the trivial dependency predicate the model-checker explores 20064 traces, and this number is reduced to 14778 with our dependency predicate.

Figure 5 shows a possible incorrect implementation of a reduce operation. The server code executes two `min` operations at steps 1 and 2 of its algorithm. Client one always sends the value 1, and client two sends the value 2. Here, the programmer incorrectly assumes that at each step the values sent by both clients will be received, and hence the minimum of the received values will be 1 in both steps. However, it could happen that in one server step the values sent during both steps of some client will be received, hence one of the assertions will be violated.

Server code

```
1  // step 1
2  val1 = receive_from( "mailbox" )
3  val2 = receive_from( "mailbox" )
4  assert( min(val1,val2) == 1)
5  // step 2
6  val1 = receive_from( "mailbox" )
7  val2 = receive_from( "mailbox" )
8  assert( min(val1,val2) == 1)
```

Code of client 1

```
1  // step 1
2  send_to( 1, "mailbox" )
3  // step 2
4  send_to( 1, "mailbox" )
```

Code of client 2

```
1  // step 1
2  send_to( 2, "mailbox" )
3  // step 2
4  send_to( 2, "mailbox" )
```

Fig. 5: Code of the Second Flawed Example.

First Counter-example

```
[server]  Recv
[client1] Send
[client3] Send
[client2] Send
[server]  Wait
[client1] Wait
[server]  Recv
[client3] Wait
[server]  Wait
[server]  Recv
[client2] Wait
[server]  Wait
```

Second Counter-example

```
[server]  Recv
[client1] Send
[server]  Wait
[client1] Wait
[server]  Recv
[client1] Send
[client2] Send
[server]  Wait
[client1] Wait
[server]  Recv
[client2] Wait
[server]  Wait
[client2] Send
[server]  Recv
[client2] Wait
[server]  Wait
```

Fig. 6: Counter-examples for the flawed algorithms

Without reductions, the model-checker tests 11562 interleavings before finding the counter-example shown in the right-hand side of figure 6. Using DPOR with our dependency predicate only 3506 traces are explored.

Finally, if we explore the entire state space, the model-checker generates 96420 traces without exploiting the independency information, in contrast to 32268 traces under DPOR.

5 Conclusion

Model checking actual distributed programs is one of the current challenges in verification, and it is feasible only if reduction algorithms such as DPOR are employed in order to reduce the number of interleavings that must be explored. Justifying the independence of actions can be subtle and should be based on a clear, preferably formal description of the operations that the system can

perform. However, realistic communication APIs for distributed programming are large and complex, and it would be a daunting task to formalize them entirely and consider all possible dependencies between their operations.

We propose instead to identify a small set of core primitives that are sufficiently expressive for encoding realistic APIs, while remaining of manageable complexity. Based on a formal specification of these primitives in TLA^+, we state and prove theorems about their independence, which ensure the soundness of the DPOR algorithm used by the model checker. Moreover, the actual implementation of different APIs in the SimGrid framework is based on these primitives, and preliminary evaluation of our verification algorithm indicates that we obtain similar reductions to state-of-the-art implementations of verification algorithms for distributed programs.

In future work we plan to validate our approach over larger distributed programs and extend it to cover more complex properties, including liveness properties.

References

1. H. Casanova, A. Legrand, and M. Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *10th IEEE International Conference on Computer Modeling and Simulation*, Mar. 2008.
2. G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.
3. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 40(1):110–121, 2005.
4. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer, New York, USA, 1996.
5. T. Hoefler, C. Siebert, and A. Lumsdaine. Group Operation Assembly Language - A Flexible Way to Express Collective Communication. In *ICPP-2009 - The 38th International Conference on Parallel Processing*. IEEE, Sep. 2009.
6. C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 179–188, New York, NY, USA, 2007. ACM.
7. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
8. L. Lamport. *Specifying Systems.* Addison-Wesley, Boston, Mass., 2002.
9. R. Palmer, G. Gopalakrishnan, and R. M. Kirby. Semantics driven dynamic partial-order reduction of mpi-based parallel programs. In *PADTAD '07: Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, pages 43–53, New York, NY, USA, 2007. ACM.
10. S. Pervez, G. Gopalakrishnan, R. M. Kirby, R. Palmer, and R. Thakur. Practical model checking method for verifying correctness of MPI programs. In *EuroPVM/MPI*, pages 344–353. Springer, 2007.
11. J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: transparent model checking of unmodified distributed systems. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association.